CS 42—Inheritance

Thursday, November 15, 2018

Summary

Today, we'll talk about inheritance: when to use it, when not to use it, and the Python syntax for inheritance.

Reusability and extensibility

Recall from last time that we'd like to write code that is **reusable** (so that other people can make use of our code, without knowing how the code works) and **extensible** (so that other people can add to our code, ideally without knowing how the code works *and* without modifying the code).

We've already seen a few ways to **reuse** code:

- **modules:** if code is in modules (i.e., libraries), we can import it
- **objects:** we build programs from self-contained objects
- **composition:** we can build a new object that *has* an existing object as an attribute

Inheritance

Inheritance gives us a way to **extend** and reuse existing code, without modifying it (and, ideally, without knowing how it works).

Inheritance enables extensibility in two ways:

- Extend for the benefit of providers: The provider of a new class can define that class by explaining how it is different from an existing class. (This kind of reuse is subclassing.)
- Extend for the benefit of clients: The client of a type can write code that can be used with multiple implementations of that type. (This kind of reuse is subtyping.)

In most OO languages, when you use inheritance, you define a subclass *and* a subtype.

Good programming practice: Prefer composition over inheritance.

Use inheritance only if the existing class and the new class have an *is-a* relationship. Otherwise, it's probably better to use composition: Instead of inheriting from an existing class, a new class should contain a field whose type is the existing class.

Inheritance in Python

When defining a subclass in Python, write then name of the superclass in parenthesis, after the name of the subclass:

```
class NewClass(ExistingClass):
   …
```

In Python, all classes implicitly inherit from a built-in class called `object`.

To call a super-class method, use `super()`:

```
class NewClass(ExistingClass):
  def __init__(self, existingAttribute, newAttribute):
    super().__init__(existingAttribute)
    self.attribute = newAttribute
```

Python classes can inherit from more than one class, but that's generally not a good idea.[1] There is one good use case of "multiple inheritance": mixins.[2]

Attribute resolution: instance, class, superclass(es)

When a running program refers to an instance's attribute, Python will try to look up the value for that attribute's name. Python always runs the same algorithm to resolve an attribute's name:

1. **Instance:** Look for a binding in the namespace of the instance. If a binding exists for the attribute's name, use the corresponding value.
2. **Class:** If resolution fails for the instance's namespace, then look for a binding in the namespace of the instance's class. If a binding exists for the attribute's name, use the corresponding value.
3. **Superclass(es):** If resolution fails for the instance's class, then look for a binding in the namespace(s) of the instance's superclass(es). If a binding exists for the attribute's name, use the corresponding value.
4. **Error:** If resolution fails for the superclass(es), throw an `AttributeError`.

[1] https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

[2] If you're familiar with Java, mixins often show up when we implement more than one interface. https://en.wikipedia.org/wiki/Mixin

Next time: static types (+ Java)